SC/1164/TS
AS

EmbeddedJava™ Technical Overview

Version 1.0

Sun Microsystems, Inc.

Sun Microsystems, Inc. (SUN) hereby grants to you at no charge a nonexclusive, nontransferable, worldwide, limited license (without the right to sublicense) under SUN's intellectual property rights that are essential to practice the EmbeddedJava Specification ("Specification") to use the Specification for internal evaluation purposes only. Other than this limited license, you acquire no right, title, or interest in or to the Specification and you shall have no right to use the Specification for productive or commercial use.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-1(a).

This software and documentation is the confidential and proprietary information of Sun Microsystems, Inc. ("Confidential Information"). You shall not disclose such Confidential Information and shall use it only in accordance with the terms of the license agreement you entered into with Sun.

TRADEMARKS

Sun, the Sun logo, Sun Microsystems, JavaSoft, JavaBeans, JDK, Java, HotJava, HotJava Views, Visual Java, Solaris, NEO, Joe, Netra, NFS, ONC, ONC+, OpenWindows, PC-NFS, EmbeddedJava, PersonalJava, SNM, SunNet Manager, Solaris sunburst design, Solstice, SunCore, SolarNet, SunWeb, Sun Workstation, The Network Is The Computer, ToolTalk, Ultra, Ultracomputing, Ultraserver, Where The Network Is Going, Sun WorkShop, XView, Java WorkShop, the Java Coffee Cup logo, and Visual Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS DOCUMENT OR ITS DERIVATIVES.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL

Please submit any comments to the following email address:

`embeddedjava-comments@java.sun.com`

By submission of comments to Sun, you grant to Sun the right to incorporate such comments in whole or in part into the document royalty-free and without restriction.

Due to the tremendous interest in the EmbeddedJava platform, Sun may not be able to respond to each submission received on the `embeddedjava-comments@java.sun.com`alias.

For further information on Intellectual Property matters, contact Sun's Legal Department:

- Trademarks, `trademarks@sun.com`
- Patents at 650-336-0069

Table of Contents

Introduction

The EmbeddedJava™ platform addresses the software needs of dedicated-purpose embedded devices. This document introduces the EmbeddedJava platform and describes how it provides many of the services associated with the Java Application Environment while fitting within the constraints of embedded devices.

The EmbeddedJava platform can be defined from several points of view.

- From the Java perspective, the EmbeddedJava platform contains a collection of configurable Java classes that have been reimplemented for memory-constrained dedicated embedded devices.

- From the perspective of an embedded device, the EmbeddedJava platform is a Java application environment that runs on top of a real-time operating system (RTOS).

- From the perspective of a Java application, the EmbeddedJava platform contains a set of configurable Java classes divided into two groups. The first group have API's that are derived from the core Java classes found in JDK 1.1. The second group contains hardware-specific classes.

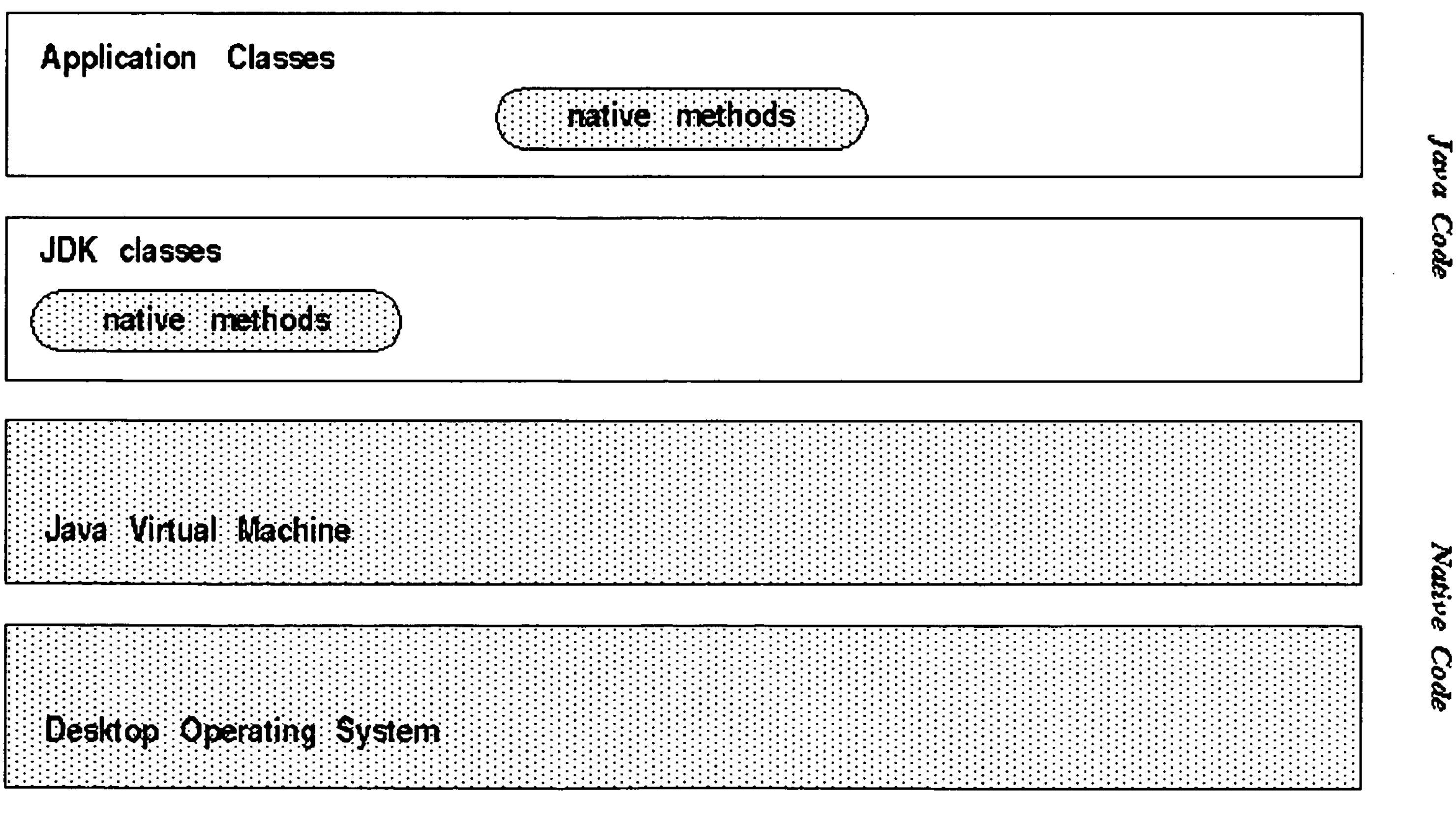
Software application design and development for the EmbeddedJava platform is different than for the desktop-oriented JDK. EmbeddedJava applications can be designed in parallel with EmbeddedJava devices.

The EmbeddedJava platform provides a licensee with a flexible way to trade off memory size against feature sets. This flexibility allows a variety of product designs from small devices with dedicated applications that are permanently stored in ROM to more complex devices that are connected to a network and can download programs and data. Example EmbeddedJava devices include low-end cell phones and test and measurement equipment.

EmbeddedJava Architecture

The EmbeddedJava architecture is similar to other Java platforms. It includes Java class libraries and the Java Virtual Machine* (VM) that runs on top of an operating environment, in this case an RTOS.

Figure 1: JDK Architecture



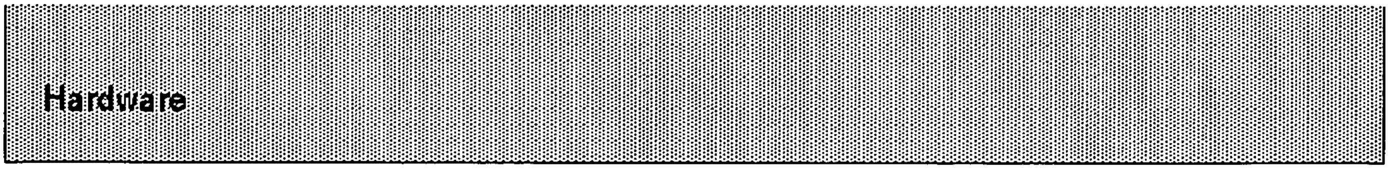


Figure 2: EmbeddedJava Architecture

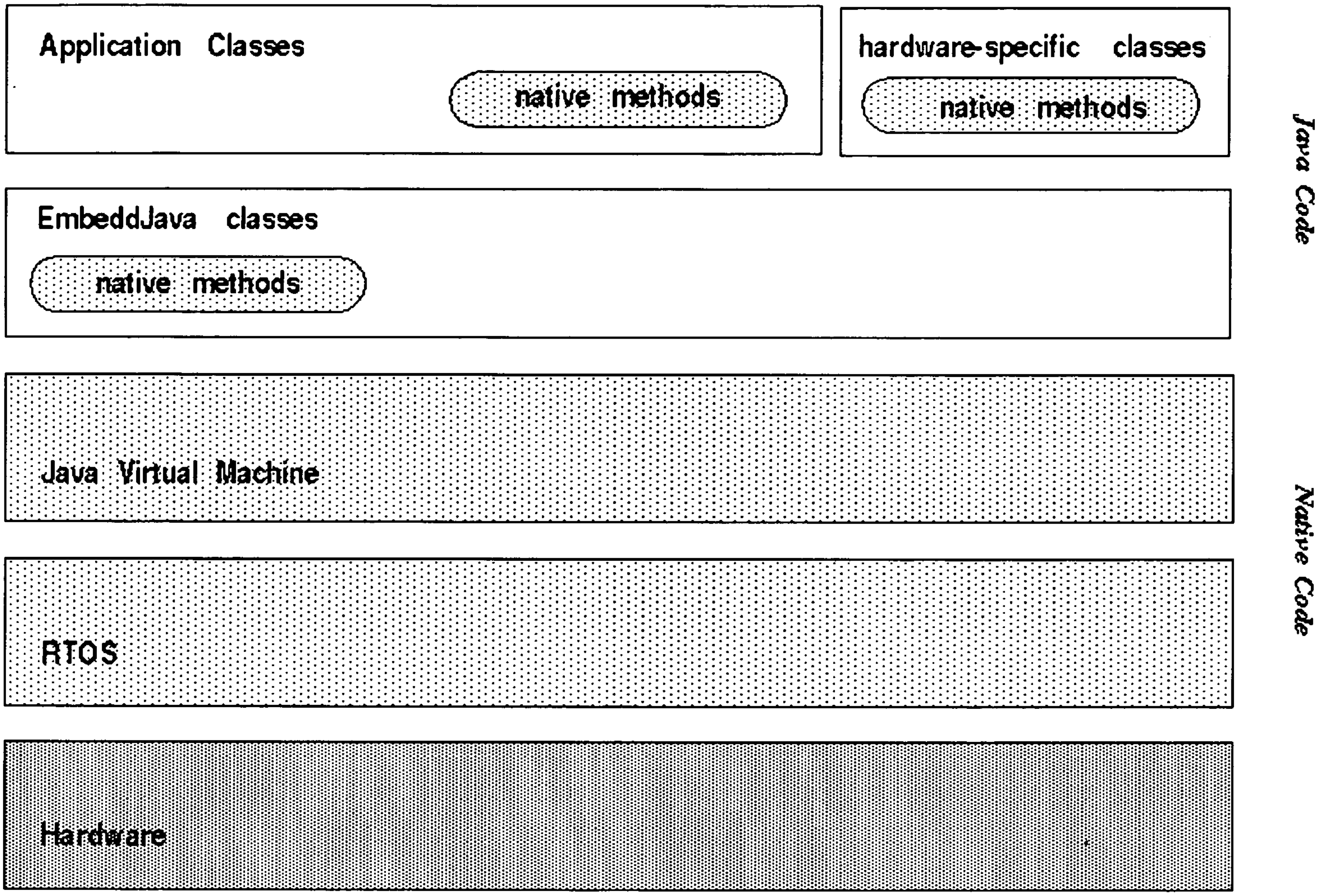


Table 1: Components of the EmbeddedJava Architecture

Component	Description
Application Classes	EmbeddedJava devices include dedicated applications that control the embedded device and perhaps manage its user interface. EmbeddedJava applications are either supplied by the device manufacturer or an ISV.

EmbeddedJava Classes	These classes have API's that are compatible with their JDK 1.1 counterparts but have implementations that have been optimized for embedded devices. The set of classes available to a licensee for building an EmbeddedJava device include all of the classes in the JDK 1.1 Java Platform Core API with the exception of the `java.applet` package. They are either supplied by JavaSoft directly or through an RTOS reseller.
Hardware-Specific Classes	These classes control the hardware in an embedded device. They are either supplied by the device manufacturer or the RTOS reseller.
Virtual Machine	The virtual machine (VM) loads Java class files, interprets bytecodes and handles native methods. The VM is either supplied by JavaSoft directly or through an RTOS reseller.
RTOS	The RTOS is supplied by a third-party RTOS vendor. At a minimum, this represents an operating system capable of supporting the EmbeddedJava VM.
Hardware	The device hardware is supplied by the device manufacturer. At a minimum, this would include a CPU, non-volatile memory for code and data storage, and volatile memory for stacks and heap storage. Additional hardware for graphics, networking or product-specific features may also be necessary.

Goals

The design effort for the EmbeddedJava platform has three goals:

- memory conservation
- customizability
- reliability

To allow the EmbeddedJava application environment to fit into the very tight memory footprints of embedded devices, JavaSoft has developed a three-pronged strategy:

- *Reimplement standard JDK classes to reduce their memory consumption.* For example, some classes in the JDK version of `java.text` allocate large local buffers because they were developed for desktop environments with less restrictive memory requirements. JavaSoft has optimized the implementation of these classes for the low-memory conditions of the EmbeddedJava platform.

- *Develop an efficient VM implementation optimized for the EmbeddedJava architecture.* The EmbeddedJava VM requires less memory and allows design-time selection of RAM/ROM tradeoff issues. It is intended to run without a JIT or virtual memory.

- *Develop tools and techniques for configuring the EmbeddedJava application environment to support specific product designs.* When a licensee starts to develop an EmbeddedJava-based product, they can choose a set of classes to include with the product. This adds a configuration phase during the design of an EmbeddedJava device. The EmbeddedJava development environment includes special tools like Java Filter and Java CodeCompact for shrinking the size of an executable image.

Dynamic class loading also provides licensees with opportunities for developing products with a rich feature set that use memory efficiently. For example, an EmbeddedJava device could allow a user to select a service and download

the classes necessary for that service across a network connection.

RTOS Requirements

The EmbeddedJava platform has a few minimum requirements of the underlying RTOS:

- memory management
- thread support
- file system (*optional*)
- graphics/window system (*optional*)
- network support (*optional*)

Definitions

For consistency, the following definitions are used for this document.

EmbeddedJava platform
A Java platform represents a set of technology and product configurations that distinguish it from other Java platforms. The EmbeddedJava platform includes a Java VM and a set of Java classes that have been optimized for dedicated-purpose embedded devices.

EmbeddedJava application environment
This is a more technical term that refers to the EmbeddedJava VM and classes.

EmbeddedJava device
A device designed by a licensee that includes hardware, an RTOS runtime system and a *configured* EmbeddedJava application environment which includes a specific set of EmbeddedJava classes that have been selected by a licensee at design-time.

ROMlet
A single executable image that contains the necessary classes and native code for an embedded device and is stored permanently in ROM or non-volitile RAM.

Patchlet
An update module for modifying the Java classes in a deployed device. This requires reprogrammable non-volatile memory such as Flash ROM or battery-backed up RAM.

Additional Reading

- EmbeddedJava API Specification
- Using JavaCheck, Version 2.0
- Java Language Specification
- Java Virtual Machine Specification
- JDK 1.1 Documentation

Configuring the EmbeddedJava Application

Environment

"Omission is the better part of compression." --Chris Sears

The EmbeddedJava platform gives licensees a great deal of flexibility in the choices and tradeoffs they can make in the design of their products. A licensee will *configure* the EmbeddedJava platform by selecting the classes and native code required to support a specific product and selecting certain VM options. Special software tools described below can streamline this process.

This section introduces configuration. The next section describes the tools and procedures provided with the EmbeddedJava development environment.

Development Cycle

From a software development perspective, there are five important times to consider during the life of an EmbeddedJava device. Each of these times reflects an opportunity for modifying the functionality or behavior of an EmbeddedJava device.

Prototype Time
: A licensee can use commercially available software development environments to prototype different elements of the application, including the user interface.

Design Time
: At design time a licensee can choose which Java classes the product will include as a permanent part of its runtime. Generally, this means selecting a set of classes that will be in the final ROM image. The software tool that assists with this procedure is *JavaConfig*.

Build Time
: After designing an EmbeddedJava device, the set of classes the product uses is known. At build time, the EmbeddedJava development environment builds a single executable image, also called a *romlet*, that contains the necessary classes and native code for an embedded device. *Java Filter* and *Java CodeCompact* are two development tools that the EmbeddedJava development environment has for optimizing the executable image to include only the code that will actually be used by the EmbeddedJava device.

Run Time
: EmbeddedJava devices are either standalone devices, in which case they cannot be modified during runtime, or they are dynamic devices in which case the EmbeddedJava VM can perform dynamic class loading.

Update Time
: The EmbeddedJava platform provides a mechanism for licensees to field upgrade their products by loading *patchlets*. These can include modifications to the EmbeddedJava VM or new or modified classes.

Dynamic Class Loading

One of the most important choices that a licensee will make in designing a product with the EmbeddedJava platform is whether to support dynamic class loading. This choice leads to two significantly different scenarios:

Supported

If a product supports dynamic class loading, then the licensee's configuration of the EmbeddedJava platform will include a VM that supports this feature. Dynamic class loading is useful for EmbeddedJava devices that are network enabled and interact with servers to download classes at runtime. At design time, the licensee must decide which classes are necessary to support the classes that are likely to be dynamically loaded.

Disabled

The EmbeddedJava VM behaves quite differently when dynamic class loading is disabled. In this case, all of the classes for the application(s) and the EmbeddedJava platform are known at build time, so the bytecode verifier may be removed from the VM. A preverification procedure may be performed on the classes and class loader included in the EmbeddedJava device. This configuration option is appropriate for conventional embedded devices that are permanently stored in ROM with a fixed application.

EmbeddedJava Development Environment

The EmbeddedJava development environment includes a set of development tools for configuring and building an EmbeddedJava application environment for a specific embedded device. The relationship between a configured application environment and an EmbeddedJava device is very close. This saves memory and optimizes performance for the needs of the device.

At design time, a licensee configures the EmbeddedJava application environment by selecting the Java classes and native code required to support the functionality of the embedded device. This selection is made from the group of EmbeddedJava classes and hardware-specific classes described in Table 1.

At build time, a licensee uses the EmbeddedJava development environment to create a *ROMlet*, a single executable image that contains the necessary classes and native code for an embedded device. In the simplest case, ROMlets can be used in devices that don't require updates. These devices can store the ROMlet in masked or one-time programmable ROM for non-volatile storage

Alternatively, an EmbeddedJava device can be modified in the field with a *patchlet* after a device has been deployed. The patchlet contains updates that supplement the initial ROMlet. Patchlets can be used in products that require updates and have reprogrammable non-volatile memory such as Flash ROM. Only the patchlet needs to be stored in reprogrammable memory, the ROMlet can reside in masked or one-time programmable ROM.

Tools

JavaConfig

JavaConfig allows a licensee to specify at design time how the VM and class files are supported by an implementation of the EmbeddedJava platform. Options can be set for removing class loading or bytecode verification. Another example is configuration of memory usage to be optimized for minimizing volatile memory or minimizing non-volatile memory.

Java Filter

Java Filter takes a list of application class files and generates a description file that identifies the all of the methods and fields required to support the application. This description file can be used by *JavaCheck* and *Java CodeCompact.*

Java CodeCompact

Java CodeCompact takes a description file generated by Java Filter and all the necessary class files and removes all unused methods and fields. It then generates an optimized C file that represents preloaded classes. This image has external references to only the native functions that are required to support the methods actually used. This tool is used at build time as part of the normal software build process.

Java DataCompact

Java DataCompact formats static data into an efficient representation that can be stored permanently with a ROMlet in an EmbeddedJava executable image.

JavaCheck

JavaCheck can be used by ISVs to determine whether their class files will run on an EmbeddedJava device. It makes this determination by checking the class files against third-party class files and a description file generated by Java Filter that represents a given EmbeddedJava configuration.

Java Mix

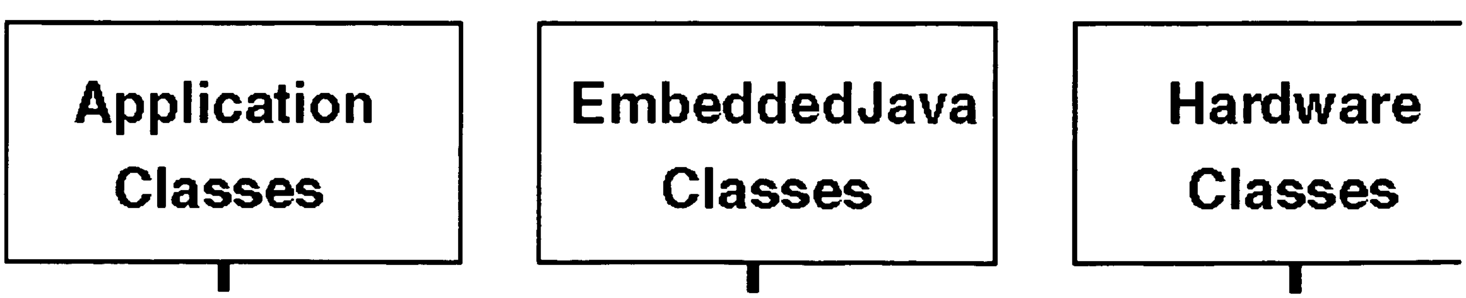
Java Mix creates patchlets from ROMlets and other patchlets.

ROMlets

Figures 3, 4 and 5 show how the EmbeddedJava development environment uses Java Filter and Java CodeCompact to build a ROMlet. Java Filter starts with the set of application classes and any platform specification files and generates an intermediate file that consists of a list of the methods and fields required to implement a platform supporting the user's application. Java CodeCompact generates a C file that contains optimized preprocessed source code for the ROMlet. This C source code file has external references to only the native functions required to support the methods actually used. It is compiled and linked with the necessary native code and the code representing the RTOS. The end result is ROMlet file that can be stored in the target device.

Java Filter determines the list of required methods and fields by analyzing the class files representing the user's application. All methods and fields determined to be required by a static dependency analysis are included in the output. The mechanism used to determine the dependencies actually analyzes the bytecodes of all required methods beginning with a `class.main()` method to determine what those methods depend on. If a field or method is not used by the bytecodes, it is not included in the output. To provide for implicit dependencies, such as with references not detectable by bytecode analysis like the use of `Class.forName()`, Java Filter can take as input a list of classes, methods, or fields that are required but not referenced.

Figure 3: Using Java Filter



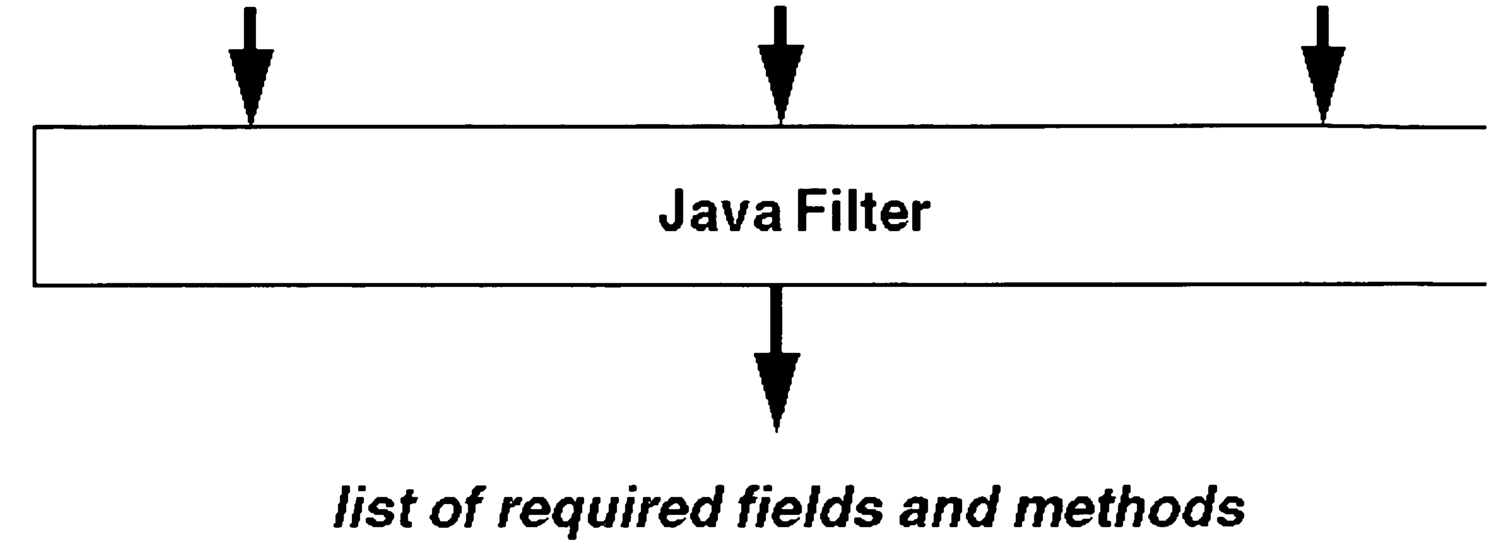
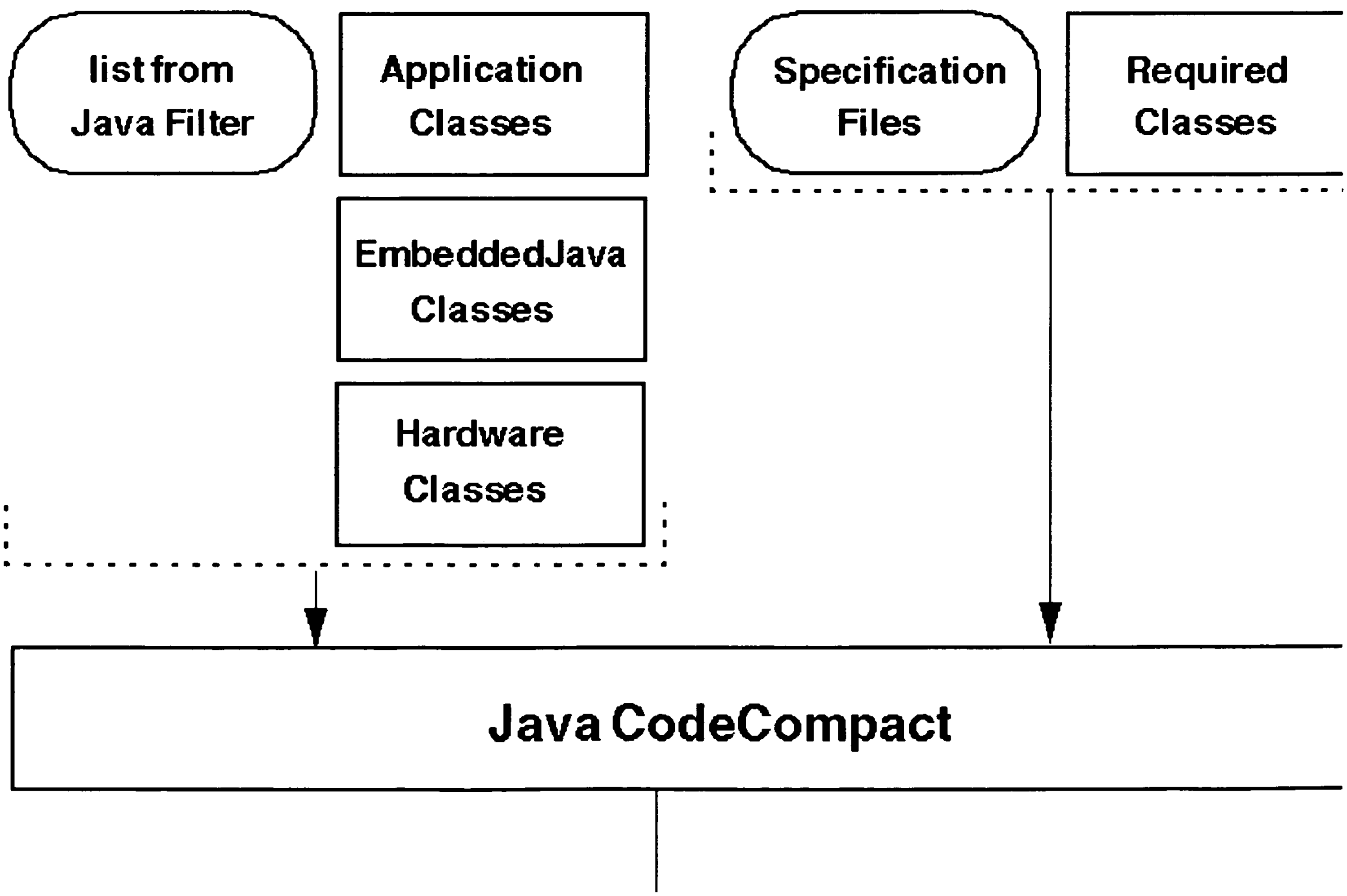


Figure 4: Using Java CodeCompact



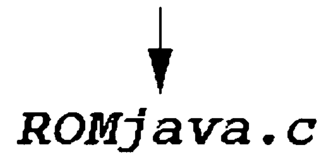
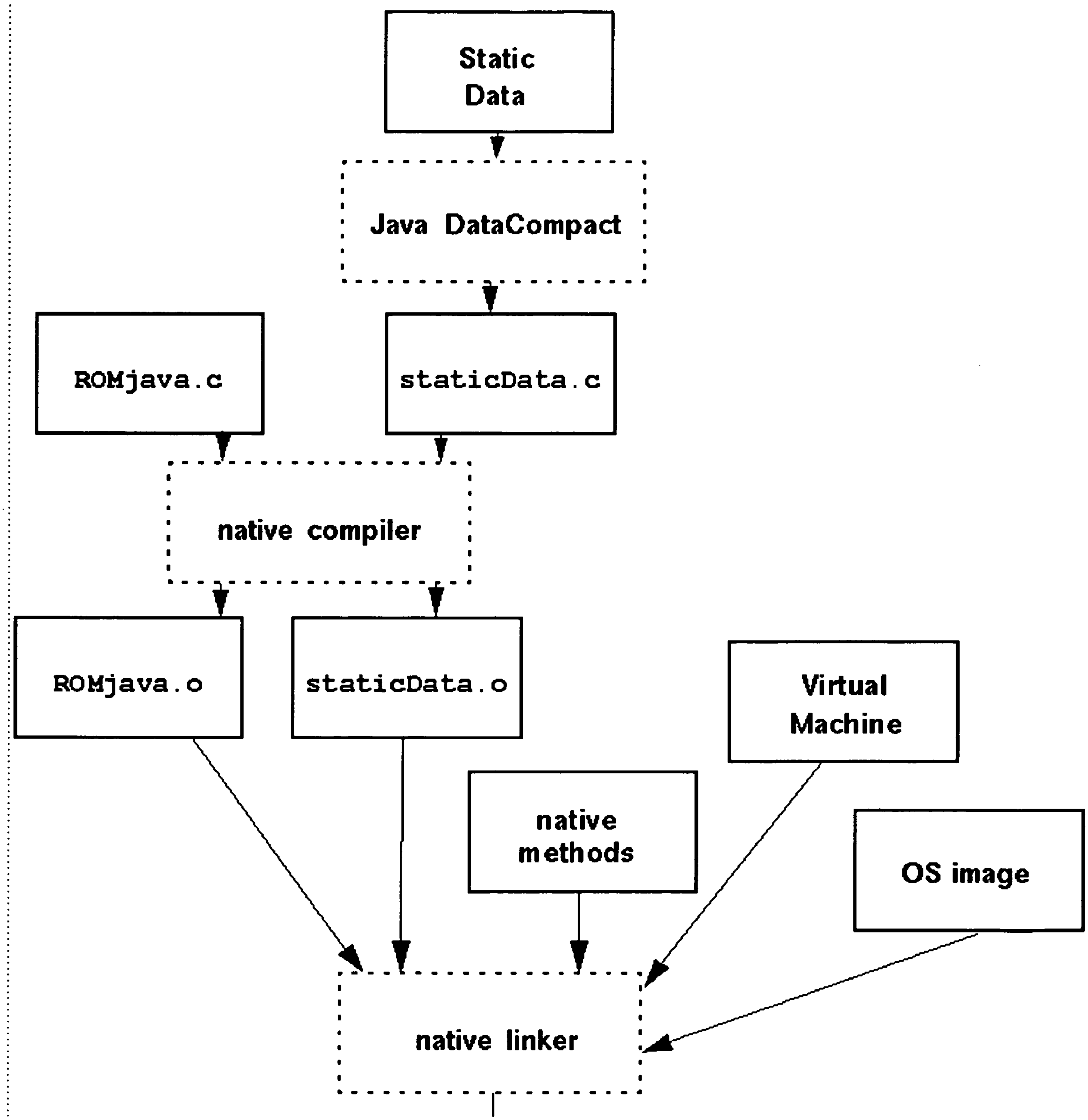


Figure 5: Building an EmbeddedJava Executable Image

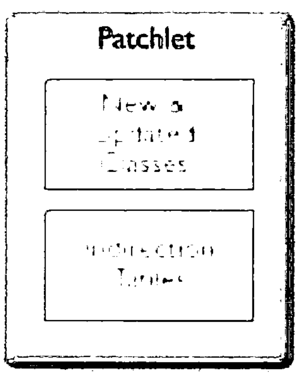


Patchlets

Patchlets allow licensees to develop embedded devices that can be updated in the field. To provide the ability to patch limited function devices, the ROMlet capability is enhanced with support for patchlets. The additional code size for this feature represents very low overhead.

A patchlet consists of two parts, a set of indirection tables that allow classes to be updated and the new and updated classes. A patch-capable embedded device has two parts: the ROMlet that defines the initial functionality and a patchlet that supplements that initial functionality by updating classes and/or adding new classes. An EmbeddedJava device ships with a patchlet that has only indirection tables that all point back to the classes in the ROMlet.

Figure 6: Contents of a Patchlet



Patchlets are typically stored in reprogrammable memory such as Flash ROM or battery-backed up RAM. Patchlets can contain bug fixes, new classes, even entire new applications for execution in the embedded device. Patchlets with various combinations of functionality can be generated "on-the-fly" by server-side applications. Functionality downloaded via a patchlet can be removed later by downloading a new patchlet that has the unnecessary functionality overridden.

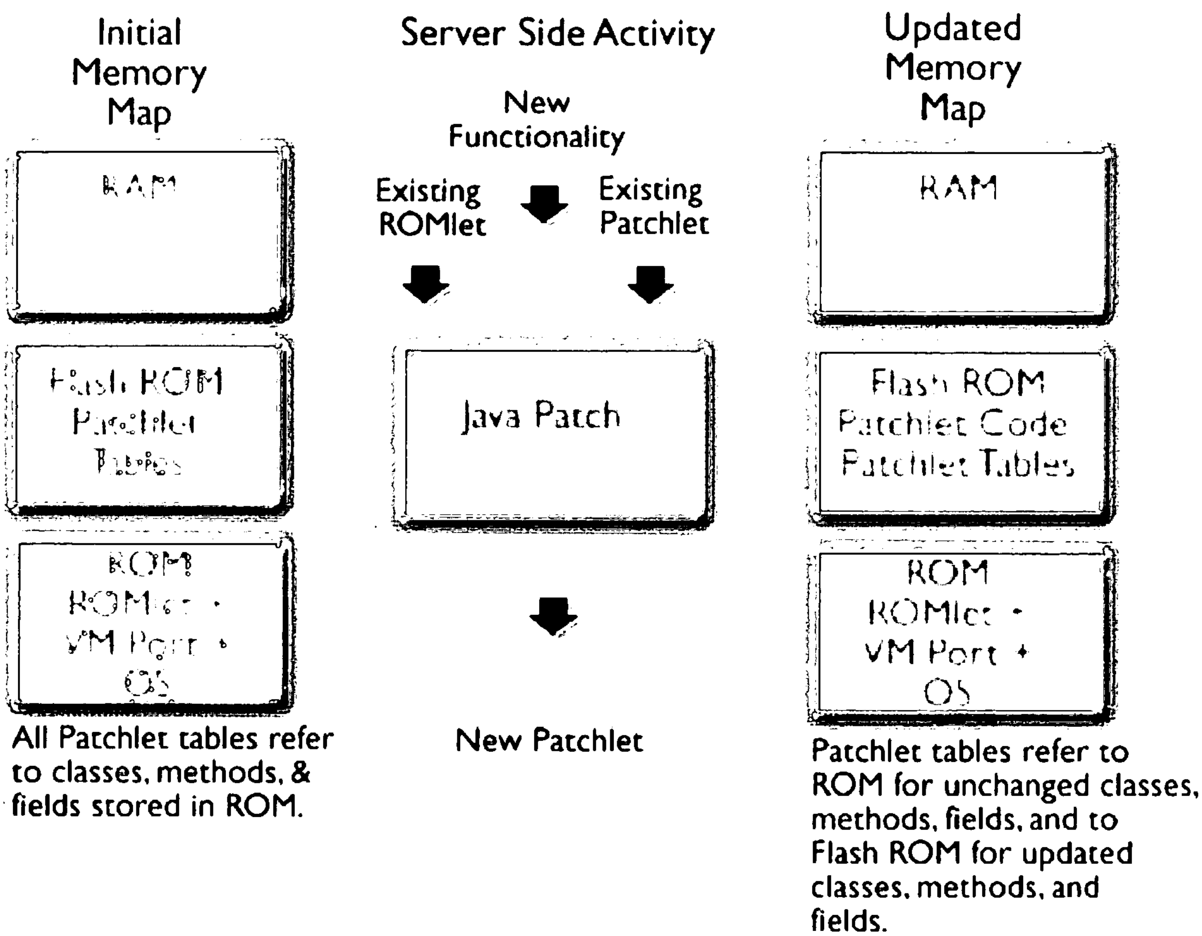
To update a system, the licensee passes the following items through Java Mix, a tool that creates a new, updated Patchlet.

- The original ROMlet
- The most recently used Patchlet
- Any new classes
- Any existing Patchlet features that should be overridden

The Patchlet is then downloaded to the target device. First the VM is terminated and then the Patchlet is downloaded. This downloading process can be done by any communication mechanism, by replacing ROM Cards, etc. After downloading is completed, the VM and associated Java applications are restarted and executed with the new functionality. This can be repeated any number of times throughout the life of the device.

Note: An EmbeddedJava device can only load a single patchlet at a time. Thus, if one patchlet needs functionality available in a previous patchlet then the earlier patchlet must be included with Java Mix when the new patchlet is built.

Figure 7: Building a Patchlet



***As used on this web site, the terms "Java virtual machine" or "JVM" mean a virtual machine for the Java platform.**